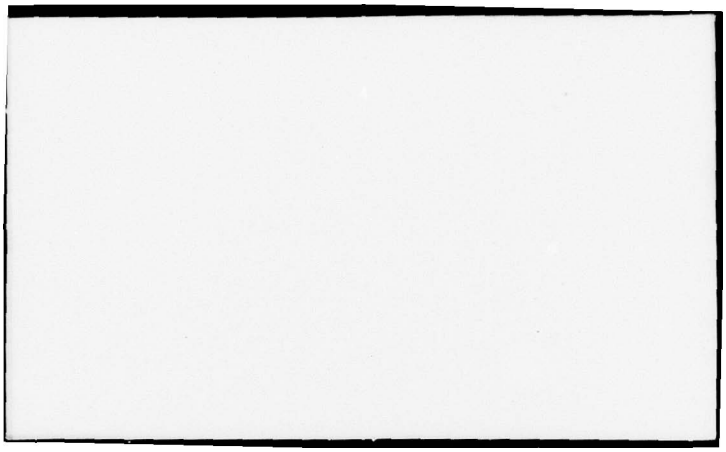
AD-A057 318 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
DYNAMIC TECHNIQUES FOR RESTRUCTURING THE CONCEPTUAL SCHEMA - AN--ETC(U)
MAY 77 E N BEAVER N00014-75-C-0462
UNCLASSIFIED 77-06-02 NL

END
DATE FILMED
9-78
DDC



DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

DYNAMIC TECHNIQUES FOR RESTRUCTURING THE CONCEPTUAL SCHEMA - AN IMPLEMENTATION

Edward Nevin Beaver

77-06-02

Contract N00014-75-C-0462

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19104

May 77

68p.

Master's thesis,

University of Pennsylvania

THE MOORE SCHOOL OF ELECTRICAL ENGINEERING

DYNAMIC TECHNIQUES FOR RESTRUCTURING THE CONCEPTUAL SCHEMA - AN IMPLEMENTATION

Edward Nevin Beaver

A thesis submitted to the Faculty of the Moore School of Electrical Engineering in partial fulfillment of the requirements for the degree of Master of Science in Engineering (for graduate work in Computer and Information Sciences).

Philadelphia, Pennsylvania

May 1977

University of Pennsylvania

MOORE SCHOOL OF ELECTRICAL ENGINEERING

Title of Thesis: Dynamic Techniques for Restructuring the Conceptual Schema - An Implementation

Abstract:

This work is a partial implementation of a dynamic restructuring processor. The total dynamic restructuring processor allows several generations of database structure to coexist. Restructuring occurs incrementally as data is accessed in the database. Only restructuring of the conceptual schema (as defined by the ANSI/SPARC report) is considered on a CODASYL type database system called WAND. The implementation uses generation data structures which allow several related schemas and databases to coexist with proper restructuring translation done on the fly. The scope of this implementation includes an analysis of data requirements and general implementation strategy for the total processor and detailed design and programming of routines that provide run-time translation from one schema definition and associated database to a user using another but related schema definition.

Degree and date of degree: Master of Science in Engineering (for graduate work in Computer and Information Sciences) May 1977.

Signed:

AUTHOR

FACULTY SUPERVISOR

1.0 INTRODUCTION

Consider the folowing sequence of events. A large medical database was implemented at some point in time with the hierachical structure as shown in Figure 1-A. This particular structure was chosen because of the one-to-many relationship between doctors and their patients that existed at that time. Several application programs were developed, one of which allowed a doctor to list his own patients.

As time progressed and the scope of the medical services expanded, patients started to be seen by several doctors rather than just one. This change recuired that the database be restructured to look like the structure shown in Figure 1-B. This change of schema structure would require the rewriting of all the doctor application programs even though the particular nature of the data relationship exploited by those programs is still present, i.e. each individual doctor still has many patients. The data in the database is of importance to the functioning of the medical facility at all times because of the need to have the data available for instantaneous lookup in case of emergency.

This example, although fictitious, illustrates many of the problems that arise when using databases and particularly large databases. The usual restructuring procedures presented to date are impractical in the situation illustrated because they cannot cope with restructuring databases that must, in total, remain on-line

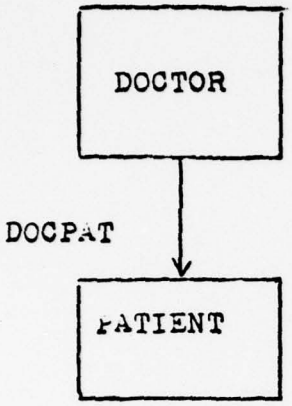


FIGURE 1-A

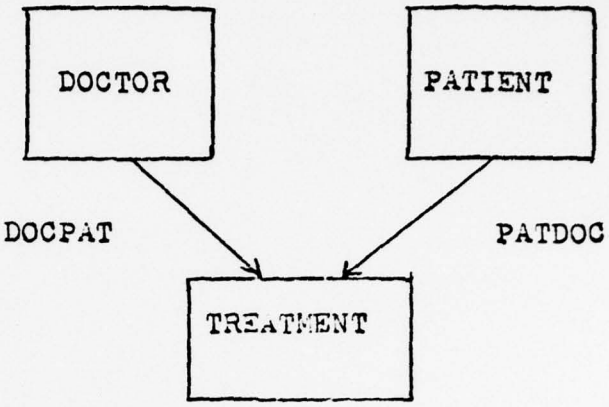


FIGURE 1-B

continuously. I call this the database continuous integrity problem. The usual restructuring procedures also do not allow old programs to remain independent of the data restructuring. This paper addresses these problems and takes steps toward the implementation of a solution.

One problem illustrated in the example and discussed in this paper is restructuring. The term restructuring has several meanings in current usage. I define restructuring to mean changing the structure of the data file necessitated because of changes in the structural or logical relationships between the data as manifested by changes to the conceptual schema.

As the example shows, databases exist in a dynamic environment and must change in order to maintain their usefulness. Database management systems and the use of sub-schema allow data file restructuring to be transparent

to applications programs to some extent. However, restructuring involving schema changes pose special problems that are addressed in this paper by the use of generation data structures [1]. Generation data structures allow applicaton programs developed with previous schema to remain independent of the restructured data as long as items required by the program are still in the database. This results in much less programing effort when changes to the schema occur. In the example, the doctor list application would not need to be rewritten after the restructuring of the schema.

The scope of this paper is limited to restructuring caused by changes in the conceptual schema. The ANSI/SPARC definition of conceptual schema is used here. The conceptual schema includes the overall definition of data items, groups of items, and the relationships between the items and groups of items in the database. The conceptual schema definition is intended to be independent of the actual stored data and is not concerned with how the data is stored.

Changes, other than those to the conceptual schema, involve data dependencies that require the restructuring to be dependent upon the value of the data actually stored, e.g. key field changes. These pose different kinds of restructuring problems not discussed in this paper. The often cited goal of data independence of database programs

naturally leads to the investigation of only conceptual schema restructuring.

The other major problem illustrated in the example is when and how to perform the restructuring. The traditional solution procedure is to take the entire database, or a partition thereof, offline and reload it under the new schema. However, in some application areas the integrity of the total database is needed at all times. This limitation makes the traditional solution procedure infeasible. Also, with very large databases, such a restructuring may take an inordinate amount of time to complete.

The solution procedure implemented here, using generation data structures, will work within this limitation by performing piece by piece or incremental restructuring. As is explained later in the paper, this type of dynamic restructuring may also be applicable to very large databases with certain characteristics.

This paper describes a procedure that performs the conceptual schema database restructuring in a dynamic manner. This procedure is limited to network structured databases (a la DBTG [2]). In addition, several steps are taken toward implementation.

2.0 PRIOR RESEARCH

A problem often grouped under the name restructuring is more aptly called reorganization. Reorganization is usually only concerned with physical management of the data, e.g. through garbage collection (physically deleting logically deleted records) [9]. Such restructuring does not apply to the work described in this paper.

The file restructuring problem, as I am defininining it, has been discussed by a number of people. Shu, Housel, and Lum [3], Merten and Fry [4], Ramirez, Rin, and Prywes [5], Burk [6], and others have developed translators for transferring data from one generation to another. But these solutions to the schema restructuring problem are handled by taking the database offline and reloading it under the new schema. These solutions work well in sequential environments where data can be taken offline for some period of time. In an online environment these solutions are harder to implement, lengthly and costly, and in some cases impossible because of database integrity restrictions. In addition, none of these solutions allow previously developed sub-schema and application programs to run, even when all the data and relationships still exist in the new data base.

Socket and Goldberg [10] give a good overview of dynamic restructuring or, as they call it, reorganization performed concurrent with usage. They give several examples where schema restructuring may be required as well as when

concurrent restructuring is needed. They give no solution procedure but rather state alternatives that exist and guidelines for the implementation.

These guidelines are : (1) The process must operate correctly , (2) Appropriate synchronization for consistency must be employed by application and restructuring processes to prevent destructive interference, (3) Deadlock must be prevented, (4) Journaling of updates and recovery functions must be included. (5) Reasonable efficiency must be provided.

The work described in this paper is an extension of the work performed by Morgan and Gerritsen [1] using generation data structures to perform dynamic database restructuring. Generation data structures allow prior schemas and their applicaton programs to operate during and after the restructuring. The restructuring as they envision it is to be performed incrementally. A data object which is described by an old schema is translated to a new schema description when the data object is next referenced by a user program.

Since this work is an extension of Morgan and Gerritsen's work, it is recommended that the reader read the referenced paper to gain a further insight and background to the work described here. In addition, their paper gives supporting arguments for some of the procedures and processes used, described, and developed in this paper.

3.0 GENERAL RESTRUCTURING PHILOSOPHY

The general method of restructuring described here is meant to be applicable to any network database system. This particular implementation is being performed as part of the WAND (Wharton Alerting Network Database) system [8]. The WAND system is an experimental subset of the CODASYL DBTG specification [2], with special features added for alerting or monitering of databases. It is implemented in FORTRAN-IV on the DECSystem-10. The complete WAND schema DDL is given in Appendix Number 4.

Figure 2 shows the structure of a medical database that is similar in part to the database discussed in the introduction. The database is a hierarchy with one branch including doctors and their patients and the other branch showing the hierarchical relationship between nurses and the hospital wings they are assigned to. The WAND DDL for this database is contained in Appendix Number 1. This example database will be used throughout this paper to help illustrate the concept of dynamic restructuring and the implementation thereof.

3.1 Restructuring Data Definition Language (RDDL)

In order to perform conceptual schema restructuring there must be some procedure whereby the changes from one conceptual schema to another can be defined. The

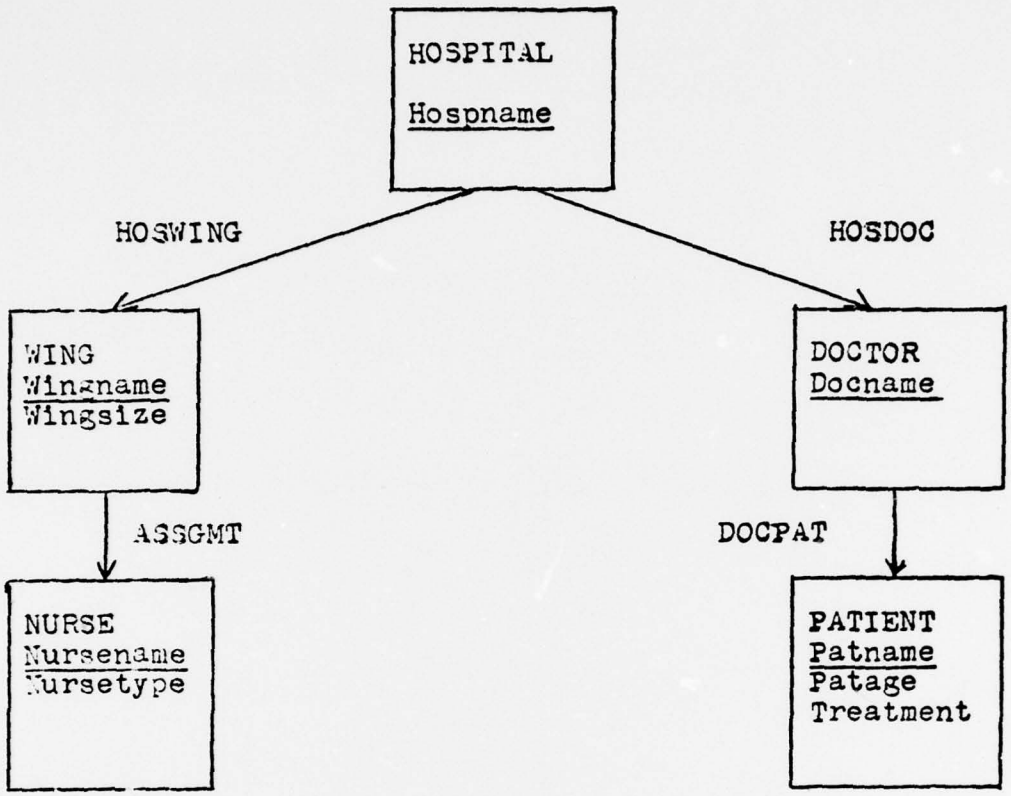


FIGURE 2

Restructuring Data Definition Language (RDDL) is the mechanism that allows the user to make such a restructuring definition.

There are many different types of definition that would be necessary go accomplish complete CODASYL restructuring. However, the scope of this work is limited to conceptual schema restructuring and any restructuring involving physical structure and physical storage in the database are therefore ignored (See Section 1.0).

Conceptual schema restructuring does not consider the following clauses since they involve physical structure: TEMPORARY area status, set MODE, MANDATORY/OPTIONAL, AUTOMATIC/MANUAL, LINKED TO OWNER, and DUPLICATES. The restructuring problems created by changing the size of an area, the page capacity, the WITHIN clauses, etc., are also physical structure changes and ignored here.

The RDDL contains three classes of entries, INCORPORATE, EXCISE, and CHANGE. INCORPORATE adds entries to the schema and EXCISE removes entries from a schema. The CHANGE entry is more complex and allows changes to individual entries of the schema. The entire RDDL is contained in Appendix Number 5.

3.1.1 Insertion And Deletion Of Schema Entries -

The INCORPORATE and EXCISE operations are handled in a straight forward manner. A simple though simple-minded procedure for restructuring schema A to schema B is to EXCISE all areas, record types, and set types from A and INCORPORATE all areas, record types, and set types that make up B. Such a procedure is not recommended but proves the completeness of the RDDL.

Whenever any particular record type is excised, all set types where that record type is the owner are also excised. If there is only one member record type of a particular set,

the excising of that record type has the effect of excising the entire set type.

3.1.2 Changing Of Schema Entries -

In many restructuring situations the restructured schema A is closely related to the original schema B. Items in the restructured schema B are usually the semantic equivalent of items in schema A. The RDDL CHANGE operator accomplishes this type of restructuring.

3.1.2.1 Change To Records -

One type of change to records is considered: changes to data items. The addition and deletion of data items are handled by the INCORPORATE and EXCISE operations. Changes of the type of the data items are easily performed and ignored here.

The only other needed change to data items is accomplished by the RELOCATE operator. This operator allows the relocation of data items from one record to another. In order to allow the restructuring processes to make this change transparent to prior schemas the path whereby the data item is relocated must be specified. In some instances there may not be a unique path so the complete specification of the path is required.

3.1.2.2 Change To Sets -

The only CHANGE clause needed for sets allows the inclusion of records as members of sets. The INCORPORATE operator allows adding records to sets, but, where several instances of the record are already included in the database before the restructuring, the RETROACTIVE SET occurrence SELECTION clause of the CHANGE operator is needed if the member is RETROACTIVE and MANDATORY/AUTOMATIC. This clause tells the restructuring routines how to assign the existing record occurrences as members of the set.

3.1.3 Restructuring Example -

Two of the most common conceptual schema restructuring changes are to convert a hierarchy of records to a confluency and to expand a hierarchy by the insertion of a record in the middle of the hierarchy. In order to illustrate the use of the RDDL, consider the medical database discussed earlier in Sections 1.0 and 3.0.

As a result of the scenario of events discussed in Section 1.0, the one-to-many relationship between doctors and patients changes to a many-to-many relationship. This necessitates the creation of a confluency. The addition of other floors to the hospital wings may make it necessary to further segregate the nurses and their assignments according to floors as well as wings. Figure 3 shows the resulting

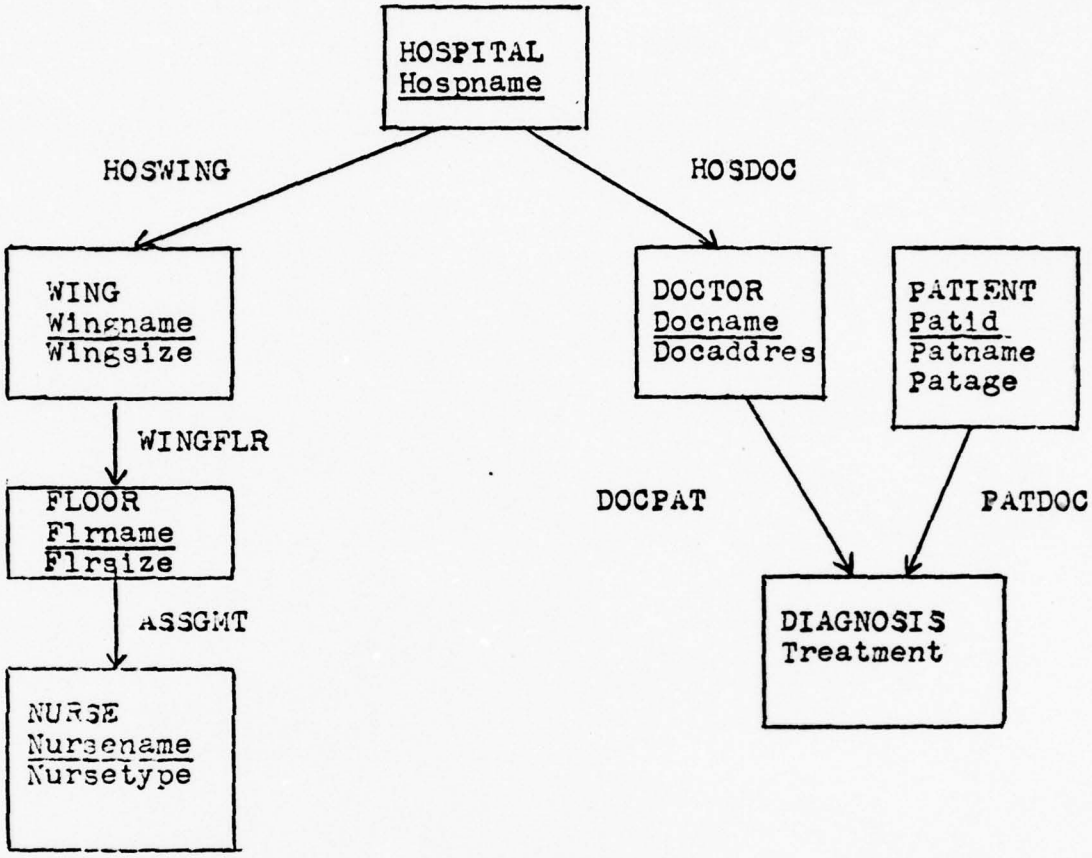


FIGURE 3

database structure that is the restructured version of the database shown in figure 2. The RDDL definition for this restructuring can be found in Appendix Number 2. The two CHANGE NAME statements are not defined but their use is obvious.

3.2 Generation Data Structures Concept

Traditionally the data in the database is stored in conformance with one schema definition with sub-schemas to support various views. Generation Data Structures (GDS) extend this concept to allow data to be stored under several

different schema at the same time. Each of these schemas being generated from a previously existing schema through the use of the RDDL. Each of these schema is considered a generation of the original schema and the different generations map the structural evolution of the database.

An application program can be written under any one of the generations of the schema. Each logical record in the database contains an indication of the schema that was in force when the record was written. Similarly, each schema generation has a generation number compiled into all its calls.

The RDDL for each generation of schema is used to develop an internal data structure that contains for each record, set, and item for every generation the information needed to perform translations into the proper schema formats on the fly. This internal data structure will be defined after a general discussion of the process by which translations are made during runtime. The RDDL is also used to develop the schema defintion for new generation.

3.3 Runtime Translation - General Description

The dynamic nature of this restructuring concept allows several generations of the schema and corresponding data to exist simultaneously. A translation occurs only as data is accessed. Two types of translation are needed: translation

of the data stored in the database and translation of the data for the run-unit so the accessing program sees the database as if it had not been restructured.

The translation is performed in a two-step process. First the data accessed is translated to conform to the latest generation schema and then translated back to look like the data from the view of the schema in force when the program was written. The reasons for this two-step translation are fully discussed in Morgan and Gerritsen's paper [1].

The translation for the run-unit is performed via the c-record and c-set concepts. The c-record (corresponding record) is defined to be that record in the current schema from which all scattered data items are accessable either because they are in the c-record or through FIND OWNER access through sets named in the "USING . . . PATH" clauses.(1) This concept is related to the VIRTUAL SOURCE concept in the DBTG report.

The c-set is a similar idea. A c-set is made up of any number (including zero) of sets in the current schema. Whenever a record is replaced by a c-record, then c-sets have to be determined for all sets of which the record is a member. The c-set for a set, X, is composed of the sets in

(1) Gerritsen, Rob and Morgan, Howard L., "Dynamic Restructuring of Databases with Generation Data Structures," Dept. of Decision Sciences, Working Paper 75-12-02, The Wharton School, December 1975, page 6.

the current schema on the unique upward path (through set owners only) from the c-record to the owner of set X.(2)

C-records and c-sets are and refer to record types and set types of the schema definition and not to particular record instances stored in the database. One and only one c-record or c-set can exist for each record or set type in the schema being restructured.

In the sample restructuring of a medical database discussed earlier, the c-record for the PATIENT record of the first generation schema would be the DIAGNOSIS record. The c-set for the DOCPAT set would be composed of only the DOCPAT set while the c-set for the ASSGNT set would be composed of both the WINGFLR and ASSGNT sets. Detailed run-unit translation using c-records and c-sets will be described in the section covering the implementation (see Section 4.0).

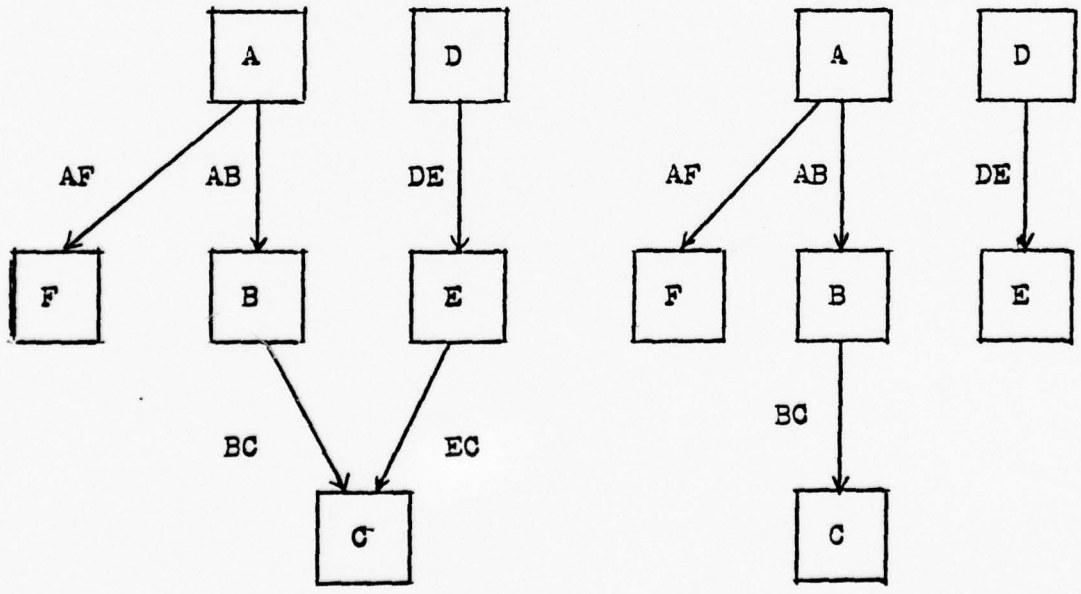
3.4 Limitations Of The RDDL And GDS

This restructuring procedure can accomplish any restructuring of schemas from one generation to another as the completeness of the RDDL indicates. However the translation back for the run-units of previous generations cannot be supported in all situations by the use of c-records or c-sets. This section will outline some of

(2) Ibid, Page 6.

these limitations. This list is not meant to be complete but rather to give an idea of the most apparent limitations.

In general, c-records and c-sets can handle expansion or reduction of access paths but not severance of access paths beyond the severance point. Expansion of access paths is shown by the medical database example developed earlier when the FLOOR record was inserted hierarchically between two other records. A reduction of an access path would occur if the sample medical database restructuring were performed in reverse order. The FLOOR and WING records in this instance would be collapsed along the WINGFLR set thus reducing the length of the access path between FLOOR records and NURSE records.



Generation 1 Generation 2

FIGURE 4

Figure 4 illustrates the severance of access paths. After restructuring to the generation 2 schema which deleted set EC, there is no longer any way to traverse an access path from records A,B,C to records D,E and vice versa. The deletion of either a record or a set in an access path will sever that path and limit access beyond the severance point.

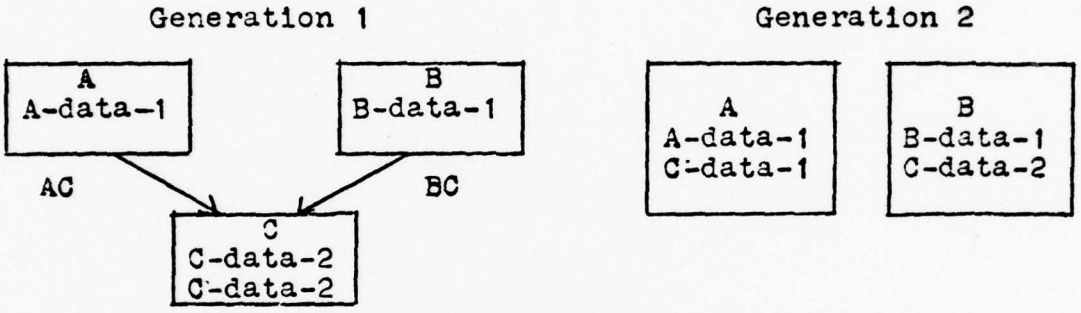


FIGURE 5

Figure 5 illustrates another limitation. Translation back to the first generation's view of record C from the second generation schema would require taking one data item from restructured records A and B. However, there is no unique c-record from which this can be performed. In addition, there is no longer an access path from record A to record C.

The RELOCATION operator allows the relocation of data items from one record to another. However, data cannot be relocated to all records in the database, it only can be located in records located hierarchically above that record for which an access path exists. For example, in figure 4 data from records B and C cannot be relocated into records D and E because an access path no longer exists. In addition,

data from records B and C cannot be relocated into record F because record F is not hierarchically above records B or C.

The reason for this limitation is that every data item must be uniquely defined by the concatenation of the keys for all the records hierarchically above it. Relocation to any of the records hierarchically above it allows the data item to be relocated in a record with a total key that is a subset of the original concatenated key. In order to relocate data items into a record not hierarchically above it requires the specification of an additional key value to uniquely specify into which particular record occurrence the data item is to be relocated. The present RDDL doesn't allow such a definition.

When an item is relocated upward in a hierarchy as the result of a restructuring, there is a problem in defining just which data item to move. If more than one record instance containing the data item exists in the database some decision must be made about which data item to relocate. This implementation will use the data value contained in the first logical record instance. It is suggested that in a refinement of the RDDL some method be devised so that this default can be changed.

The final limitation perceived is that when a data item in one generation is deleted but is converted into a logical link between records, there is presently no manner by which the restructuring processor can recreate the data item from

the logical link.

3.5 GDS Internal Data Structure

As described earlier, the translations performed by the restructuring processor use an internal database. The database used in this implementation is implemented as a WAND database. This database cannot be restructured because it is accessed by the restructuring routines. These accesses must be made with routines not using the database itself, i.e. the WAND routines developed for databases not allowing restructuring. The structure of this database is shown in figure 6 and the DDL is contained in Appendix number 3.

The SCHMA record is accessed by the schema name, NAMSCH, and the number of generations currently existing is contained in NUMGEN. For each generation of the named schema a GENCHG record is stored. This record, accessed via the value of the generation number, GENNUM, contains the name of the file where the schema definition of the generation is stored (*.SCH file). For each generation of the database the information contained in the other records gives the information needed to convert the run-unit's data from the most recent schema generation to the view of the application program's schema generation.

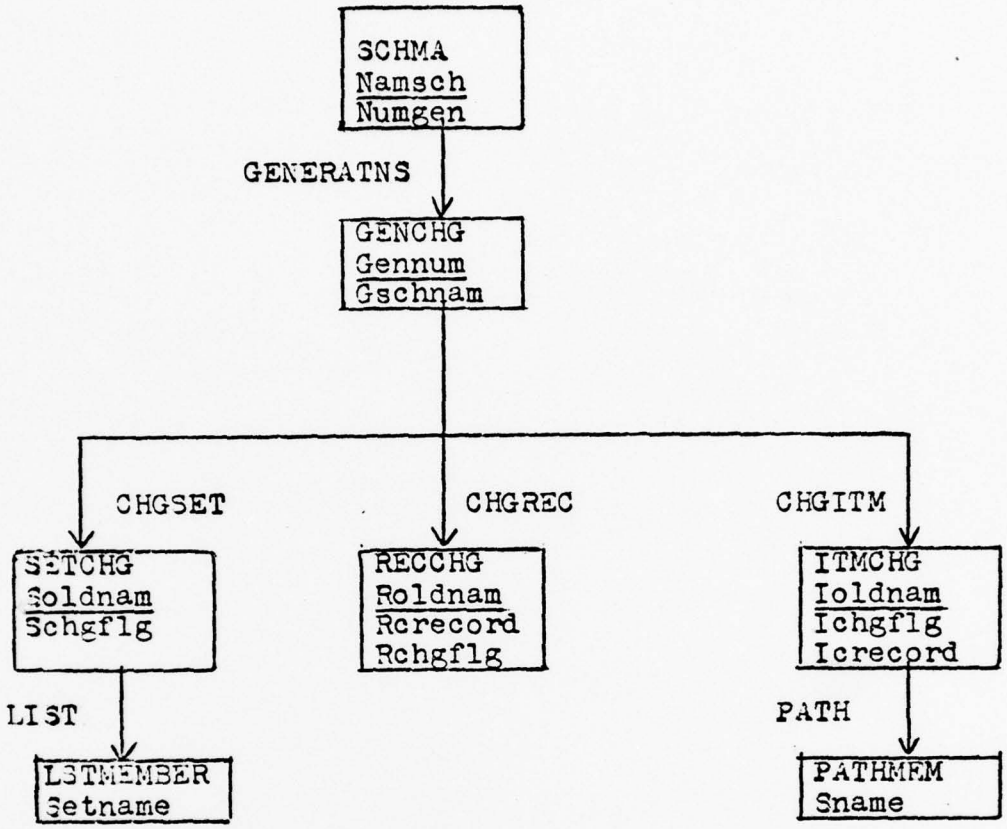


FIGURE 6

The SETCHG record, accessed via the set's name, SOLDNAM, contains a flag telling whether the set was deleted or not. The LIST set and the set names in the LSTMEMBER records associated with this record make up the c-set for the named set. The RECCHG record includes information about records. This record accessed by the record's name, ROLDNAM, contains a flag telling whether the record has been deleted or not as well as the c-record for this record.

The ITMCHG record in this internal database contains information needed to perform the run-unit translation. This record, accessed via the data item's name, IOLDNAME, contains a flag telling whether the item has been deleted or

relocated. The associated PATH set and PATHMEM records for each item is the list of sets that must be accessed to get to the relocated data item. For the SETCHG, RECCHG, and ITMCHG records, if a record occurrence does not exist in the database for a given record, set, or item name, then no runtime translation is needed for that record, set, or item.

3.6 Sequence Of Operations In Restructuring A Schema

This section will define the typical sequence of operations whereby the user goes about restructuring a schema. The original schema file is created in the normal manner as described in the WAND manual [8]. To restructure this database the user creates an RDDL file giving the restructuring definition. This file should have the extention .RDD.

This RDD file and the previous schema file (*.DDL) is inputted into an RDDL File Description Processor (RDDFDP). This processor using the inputted files creates the schema definition file (*.DDL) for the restructured schema. This file is used by applications programmers to create their programs. In addition the RDDFDP processor creates a *.SCH file which gives the internal schema representation used by the WAND DBMS. The file name for these generated files is the same name as the original schema file name with the generation number appended, or if that creates a file name that is of length greater than six characters, the sixth

character is substituted with the schema generation number.

The RDDFDP also inputs into the Generation Data Structures' internal database the information needed to perform the restructuring translation. The internal name for the schema is the same for all generations, i.e. the original schema name. Since each restructuring redefines the most recent generation schema, all the previously stored translation data must be updated. If errors occur during the RDDFDP processing a file with the extention .ERR is produced and no new .SCH or .DDL file is generated.

As an example, if the original schema was defined in a file named MEDICA.DDL and the restructuring definition was defined in a file named CHANGE.RDD the RDDFDP would create two files named MEDIC2.DDL and MEDIC2.SCH along with changing the internal database.

These new files along with the changed internal database allow the DML processors to make the necessary restructuring changes. New application programs can now be written using the newly defined schema DDL as the database definition. All other restructuring generations of the schema are created and run in a similar manner. Note that each application program can be written using only one generation schema definition.

This particular implementation only allows four generations of the database to coexist at the same time. The number four was chosen to allow sufficient generations of the schema for complete systems testing and to minimize storage requirements. The optimum number of generations allowed to co-exist is a function of several variables including the expected number of generations existing simultaneously at any point in time, size of the database, storage requirements, availability of other restructuring processors, etc. The coexistence of many more than four generations of the database could cause severe performance degradation and a total offline restructuring would probably be cost effective.

4.0 IMPLEMENTATION DETAILS

This section of the paper will discuss in more detail the implementation of the dynamic restructuring ideas described in the earlier sections of this paper. A total implementation design of the internal processes needed to perform all the restructuring translations is presented. A step by step implementation plan to implement and thoroughly test the restructuring process is then discussed. Following that will be a detailed discussion of the changes made to the present WAND system and those restructuring processes written and tested for this work. Version B00 of WAND as it existed on April 15,1977 was modified to implement the

restructuring processes.

The short time limit placed on this project precludes the writing and testing of all processes needed for a total restructuring system. But the work done should give guidance in continuing the implementation and show the feasibility of the restructuring concept discussed earlier.

4.1 General Implementation Strategy

The implementation of dynamic restructuring discussed requires three different types of processes. The three types of processes are called reconstruct, locate, and translate. Any DML call made by the user using the most recent generation schema or any other generation schema is first reconstructued to look like the call or calls the user would have made had he been using the most recent generation schema. Of course, if the user is actually using the most recent generation schema then no actual reconstruction of the call is necessary. The reconstruction is accomplished via the use of the internal database discussed earlier.

Once the call has been reconstructed the database is accessed using this call or calls in the form needed to access the most recent generation database. The data that is the target of this call or calls is stored in either the most recent schema generation database or any of the other generation databases. The locate process performs the

searching for the target data in the many coexistent database structures, i.e. the most recent generation database and all previous generation databases. The most recent generation database search is performed using the actual reconstructed call. The search in all the other generation databases is achieved by performing another reconstruction on the call. This reconstruction is in many ways the inverse of the reconstruction done on the original user's call.

There is a separate reconstructor and locator routine for each user interface routine (DML command) that is supportec after restructuring. Each user interface routine that is supported after restructuring requires different forms of reconstruction and location, thus requiring the separate routines.

Once the target data is located it must be translated into the view of the most recent generation schema. This translation is only done for target data found in databases other than the most recent generation database. The translaton is only done for the target data, all other data remains unchanged.

4.2 Implementaion And Testing Plan

I have developed a four step implementation plan that will allow for step wise or incremental implementation.

This plan also permits full testing of each phase of the implementation before proceeding onto the next step. The four implementation steps are as follows:

1) Add the needed changes to the present WAND system in order to allow the restructuring processes to operate. The restructuring processors need additional data variables and status variables. The schema definitions for all the schema generations are needed as well as user work areas for all the databases coexisting (one for each generation of the schema) in addition to the main user work area the user interacts with. These changes are tested by running the database with only one schema generation. Since only one generation exists at this stage no restructuring of calls, location of target data, or translation of data is needed.

2) Once the first step has been implemented and tested, the reconstruction processes are then implemented. No other processes are implemented at this stage. The programming of the reconstructor routines is tested with a database stored totally under the view of the most recent generation schema. Since this is the only existing database, the reconstructed calls can be made directly to the database and no locating or subsequent translation is necessary.

3) The next implementation step is to implement the locator processes. These processes are then tested with several coexistent databases stored under past generations of the schema. However, these databases remain static and

the data is not translated to the most recent generation database.

4) The fourth and final step is to add the translator routines to the database. The resulting programs will comprise a complete dynamic restructuring procedure and the full system testing can be performed.

Once these four steps have been completed any other modifications and enhancements can be added to the system. The present detailed implemention only covers steps 1 and 2.

4.3 Needed Data And Changes To Existing DML System

This section will describe the additional data that is needed by the system in order to allow restructuring. This additional data also requires some changes to the existing DML system and these changes are described.

Before detailing the additional data and changes, we must define the concept of 'context switching'. This implementation uses the WAND DML user interface routines, as they existed before this restructuring work, to make all database accesses. These routines assume that only one schema definition and user work area exists. In order to use these routines it is required that only one generation of the schema definition and one user work area be in force when making actual database accesses. The schema generation and user work area in force defines the context that exists

at any point in time.

Dynamic restructuring requires performing many different basic database accesses under the context of any of the generations of the schema as well as the user's generation of the schema. There must be a way to move from one context to another and back during the execution of a restructuring. This movement is termed context switching.

Since restructuring requires access to the databases existing for all generations of the schema, the schema definitions for all the generations of the schema must be stored. Access to all these existing databases requires a separate user work area for each database so the restructuring processes can interact with these databases. In addition, the main user work area, where the original user interacts with the database as a whole, must be maintained.

Each context swith requires a change in the generation schema in force as well as a change in the user work area being used. Therefore, in addition to the main user work area now used by the WAND implementation and the normal user schema initialization, the restructuring processes require the initialization of additional user work areas for all generations of the schema as well as schema definitions for all the generations. The method of data storage and context switching used causes some data redundancy, i.e. the user's generation of the schema definition is stored twice, once

for use by the user and his database calls, and once for use by the restructuring processor to perform its database calls to the database stored under that generation schema.

In order to allow context switching, the address bases of the location where the schema definition data is stored is now a vector of bases rather than a scalar. The address bases for the location of the many user work areas are also stored as a vector of numbers rather than the scalar value used by the present WAND implementation. These internal variable changes require the rewriting of all statements referring to locations in the schema definition storage area or the user work area.

The DBOPEN routine, that initializes the schema definition, is changed to read in all needed generation schema definitions as well as setting up all the user work areas. The DBOPEN routine also initializes the additional restructuring parameters described below. This initialization requires the opening and reading of the internal database used to store information for the restructuring, as was described earlier.

The restructuring processes require some additional system parameters which are stored in a newly created common area called RCCOM. These parameters and their definitions are as follows: SCHNUM - the generation of the schema now in force (the present schema generation in context). This variable if equal to 1 means the user's original schema

context, any other value implies a context of some generation of the schema, UWABAS - this is a vector of values telling the address base of the various user work areas, CUWABAS - a scalar which at all times contains the value of the user work area address base for the current context, RCSTACK - this vector of values is used as a stack to allow the switching of contexts, STKTOP - the scalar pointing to the last location in RCSTACK that is used, RCSWITCH - this is a switch telling the internal processes whether restructuring is called for or not. This switch is created by looking at the internally stored generation number of the schema the user is using. If this generation number is zero, no restructuring is called for, otherwise the restructuring processes are used, NUMOFGEN - this scalar contains the total number of generations of this schema that exist.

As was discussed earlier, the internal database is implemented as a WAND database. This is accomplished by appending this database, named CHGDB (Change DataBase), as a separate area of each schema generation. This area, not seen by the user, is appended as the first area of the schema definition for all generations of the schema.

The reasons for this particular implementation stratety are as follows. The internal database contains information needed to fully initialize the internal data structures, namely the number of schema generations and names of the

files containing the schema definitions for each generation. Because this data is stored in the internal database, the initialization routines need to access the internal database after inputting only one schema definition, i.e. the user's schema definition. Therefore, the internal database definition must be appended to all schema generation definitions.

CHGDB is the first area of each generation's schema definition so that all internal routines accessing CHGDB will know where in the user work area to look for the data no matter what the generation of the schema may be. In order to assure that the correct database currencies are used, only one of the internal user work areas is used (i.e. only one context is used), in making access to the CHGDB. The main user work area is the area actually used.

The reconstruction processors internally perform DML calls as a result of the user's global DML call. However, FORTRAN-IV on the DECSystem-10 does not allow the calling of a subroutine from within itself. In order to solve this problem the user does not directly call the DML routines but calls controlling routines. All the old DML routines have been renamed by adding the prefix DBX to the abbreviation of the replaced DML command subroutine. For example, the old routine FINDAP was renamed DBXFAP.

New routines were created with the old DML command names which act as controlling routines. These routines check to see if restructuring is needed, if not the appropriate DBX routine is called. If restructuring is called for, the appropritate DBR routine is called to reconstruct the DML command. If restructuring is needed but the command itself need not be reconstructed, as is the case when the user is using the most recent generation schema, the appropriate locator or DBL routine is called. The locator or DBL and DBR routines when accessing databases use the appropriate DBX routines. A DBX, DBR, and DBL routine exists for each user interface routine supported after reestructuring.

4.4 Utility Routines

There are a number of utility routines that make the programming effort easier and also provide general routines that can be used by further implementation efforts. These routines are of two types, those needed to perform context switching and those that access the internal database CHGDB.

There are two context switching routines and two routines that support those routines. One routine, called DBCTTO, will switch the context to the given schema generation number. The prior context is pushed on the stack called RCSTACK by a general stack push routine named DBPUSH. The needed conversion of the context and context bases is

then made.

The other context switching routine, called DBCTBAC, switches the context back to the previous context. A general stack pop routine, called DBPOP, is used to get the previous context off of the stack named RCSTACK. In order to switch the context two internal parameters are set to the proper value, i.e. SCHNUM and CUWABAS discussed earlier.

All access to the internal database CHGDB (see Figure 6 and Appendix Number 3) is through utility routines. The internal database is initially entered through the routine DBGINIT. This routine reads the SCHMA record by using the inputted schema name and returns the number of generations that exist of that schema. The routine DBNXSCH finds and reads the GENCHG record using the inputted generation number and returns the file name where that generation's schema definition is stored.

The routines named DBDSET, DBDREC, and DBDITM read the records named SETCHG, RECCHG, and ITMCHG respectively. The particular records are located by the use of the inputted set, record, or item name. The data in the accessed record is returned by these routines along with any error status that was produced as a result of these calls. The error status once gotten from the user work area is reset to zero to allow further processing. The routines calling the DBD routines manage these errors separately. Some errors are expected as a result of the internal database accesses,

however, all unexpected errors will eventually be returned to the user. All of these routines assume that the correct GENCNG record is current for making these calls.

The final two utility routines that read the internal database are DBDPATH and DBDLIST which access the PATHMEM and LSTMEMBER records and the set that owns these records. The PATH and LIST sets are ordered sets of setnames that are accessed in some sequence to reconstruct DML calls. These routines input the position within the sets PATH and LIST where the needed record is located. The resulting set name gotten from the PATHMEM or LSTMEMBER records is returned along with any error status. The error status is handled like the other DBD routines handle it. Both routines assume that the appropriate owner record For the PATH and LIST sets is current.

4.5 Detailed Reconstruction Routines

This section will discuss in detail all of the DML commands that reconstruct and how that reconstruction is accomplished.

4.5.1 The GET Command -

The GET command moves the current instance of the indicated record type from the database into the user work area. The reconstructed GET command does essentially the

same thing.

The internal database is checked to see if the named record has been restructured. If the record type has been deleted, signified by a particular flag value, an error is returned (see Appendix Number 6). After switching to the most recent generation context and updating the currency of the record type, the database is accessed. The named record type or the c-record, if it exists, is the actual record type gotten from the database through the use of a DBLGET call.

Now that the proper record has been gotten from the database, the data must be loaded into the main user work area. This is done by sequencing through all data items contained in the record according to the user's schema generation. For each data item the internal database is accessed to see if the data item has been restructured. If no restructuring occurred, the data item is moved directly from the last user work area, where the data from the database now rests, to the main user work area. If the data item has been deleted, a null value is moved into the main user work area locations for this data item. This convention is used by the CODASYL committee [2] in the ACTUAL/VIRTUAL SOURCE specification. In that specification if the data value does not exist a null value is returned. This convention therefore seems appropriate for signifying that no data value exists rather than returning an error.

If the data item has not been deleted or is not in the record obtained from the database, it must be relocated in some other record. To get this record FIND OWNER accesses are made for all sets in the relocation path stored in the internal database. The record thus located is moved from the database into the last user work area and the appropriate data is moved to the main user work area.

After all data has been moved, the main user work area currency is reset to the database key of the record found in the database as a result of the original database access (i.e. the record or its c-record, if it exists). Note that all database accesses except those to the internal database, are done by DBL rutines that locate the needed data in the database. In addition, any unexpected errors that occur as a result of a database accesses are returned to the user.

4.5.2 DELETE Command -

DELETE is similar to the GET command recontruction. The internal database is accessed to see if the record to be deleted has been restructured. If the record no longer exists in the schema, an error is returned (see Appendix Number 6). Otherwise, the record as named (or its c-record, if it exists) is deleted from the database. The particular record to be deleted is signified by the current database key found in the main user work area for the record type originally specified. After the deletion has occurred, that

currency in the main user work area is reset.

The delete command deletes the current instance of the record or its c-record and all records linked beneath that record. It is possible for a person using a prior generation schema to delete records, or data values stored therein, that he does not know exist and that may be used by other users who are using more recent generation schema definitions.

4.5.3 STORE Command -

The STORE command cannot be reconstructed. Consider the restructuring illustrated in Figure 7 where the only change from generation one to two has been the addition of a record and a set to complete a confluency. When a program written with the first generation schema tries to store an instance of RECORD-2, the restructuring now requires that some record of type RECORD-3 be selected to establish the appropriate links. However, there is no way of selecting that record because it did not exist under generation one and data item DATA-3, which is the record's key, does not exist for programs written with generation one schema.

In general, to store a record, the full concatenated key for records above the record to be stored must be specified in some way. If restructuring changes a record's identifying key by additions, deletions or changes to keys,

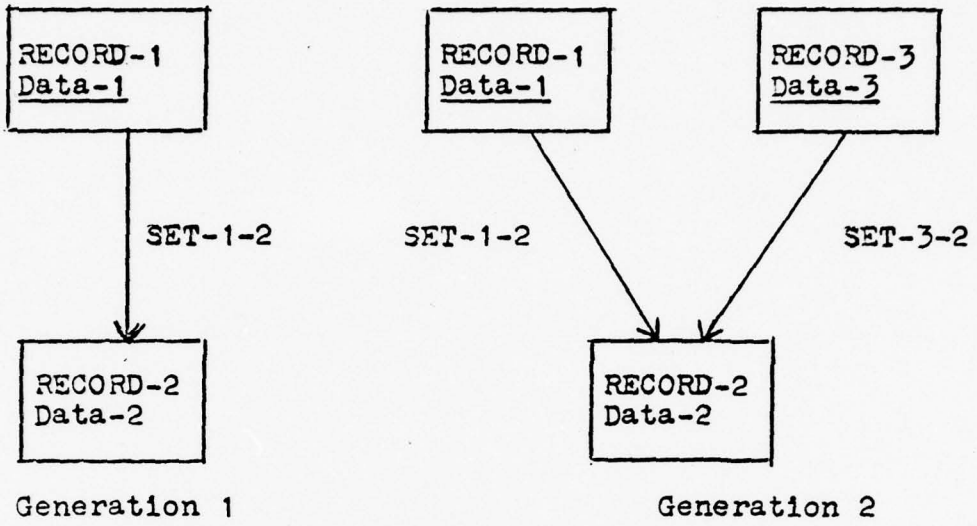


FIGURE 7

the STORE cannot be adequately reconstructed. For these reasons the STORE command cannot be reconstructed and any call to this routine in any generation but the most recent generation will cause an error (see Appendix Number 6). It is recommended that all storing in a database that has been restructured be by routines written using the most recent generation schema definition.

An alternative, and possibly better solution, is to allow programs using previous generation schema to STORE a record type if it can be verified that the related structure has not been changed. The implementation of this alternative would require additional data be stored in the internal database to signify whether a record can be STORED as a result of the restructuring.

4.5.4 MODIFY Command -

Unlike the STORE command, the MODIFY command can be reconstructed because modifications are performed on records that have been located in the database previous to the MODIFY command call.

The MODIFY command is reconstructed in the following manner. The internal database is searched to see if the record type has been changed by restructuring. If the record type has been deleted, a restructuring error is returned (see Appendix Number 6). If a c-record exists for the record type, the c-record will be used as the base record to make all modifications rather than the record type originally specified because all data items for the specified record are located in the database with respect to the c-record.

Because the record or c-record to be modified may have more data in it than the user is aware of, the modification must be made to the currently stored record or records that contain, after restructuring, the data items in the record that the user wants to modify. The current record or c-record is first gotten from the database. For every data item contained in the record the user wants to modify, the following steps are performed. If the data item no longer exists due to restructuring, it is ignored. If the data item has been relocated, it is stored on a stack for later processing. All remaining data, i.e. those items that

after restructuring are still contained in the record or its c-record, is moved from the main user work area to the correct locations in the last user work area where the data values obtained from the database lie.

After all data items have been sequenced through, a modification command is performed to place the modified record back into the database. For all data items relocated in other records, the following operations are performed. FIND OWNER commands for all sets in the relocation path (the path is stored in the internal database) are made to locate the record that has a data value to be modified. The located record is gotten from the database and the data item value moved from the main user work area to the last user work area. The record is then reentered into the database by a MODIFY command.

The modification should update the database so that the newly stored data item values after the modification can be gotten back as entered. Therefore, the processing for the relocated data is performed after the wanted record or its c-record has been modified. That modification may change some set relationships and thereby change the particular record in which the relocated data now is supposed to lie.

The current record instance that is to be modified is determined from currency indicators in the main user work area and after modification these currency indicators are updated. Any errors that may have occurred as a result of

the database accesses are returned to the user.

4.5.5 FINDAP Command -

If a record is named in the calling portion of this command, the internal database is searched to see if the record has been restructured. If the record has been deleted, an error is returned (see Appendix Number 6). If the record still exists or has a c-record, the database is accessed using a DBLFAP command using the record named or its c-record, if it exists.

If no record is named and 'ANY' record is requested, the following is done. The most recent generation database is searched for any record using the FINDAP location routine. If the record found exists in the user's generation database, that record is returned to the user. The database is searched until such a record is found or an error occurs. This procedure will only find and return to the user records that have not been restructured. If it is deemed necessary to return any record that the user would normally expect to find, it would be necessary to check each time a record is found to see if it is a record in the user's schema or if it is a c-record for a record in the user's schema. This requires a search of the internal database and is thought presently to be unnecessary for this command since the command's use is thought to be rare.

No matter how the record is found in the database, currency is updated in the main user work area using the database key for the record found and any errors that occur are returned to the user.

4.5.6 FINDC Command -

As with all other commands involving records, the routine to restructure the FINDC command accesses the internal database to see if the named record has been restructured. If the record has been deleted from the most recent generation schema, an error message is returned (see Appendix Number 6). If the record has been restructured and a calc access is impossible, as is the case if the calc key has been deleted or relocated in some other record besides the c-record, another error status is returned (See Appendix Number 6).

If the record now has a c-record where the calc key resides, special processing has to be done to simulate the calc access. Because the c-record contains the key but is not stored via the key value, all c-records must be sequentially searched until a match on the key value is obtained. In order to allow for the finding of duplicates, the first sequential access of the c-record is done using the position as entered, i.e. 'FIRST' or 'NEXT'. This causes the search to start at the beginning of the database or where the last search left off depending respectively

upon whether the first or the next duplicate is wanted. Although this method is not foolproof, it allows, in most cases, the finding of duplicates and is better than no provision for finding duplicates, however, it can be expensive if many records of the requested type exist in the database.

If the record has not been restructured, the calc access to the database can be made in the usual manner. When the wanted record is found, regardless of how it was found, the database key is returned to the main user work area and currency is updated. All errors encountered are returned to the user as he would see them under normal operating conditions.

4.5.7 FINDPO Command -

The FINDPO command operates on sets. The internal database is accessed first to see if the set has been restructured. If no restructuring was done on the set the command is used directly to locate the target record in the database. The current position in the set is found by the currency indicated in the main user work area which is transferred to the last user work area where the database access is made. If any error occurs it is returned to the user. The currency in the main user work area is also updated after the database access.

If the set has been deleted an appropriate error is returned (see Appendix Number 6). If the set has been restructured and not deleted a c-set exists in the internal database for the set. The processing to be done to reconstruct the command is dependent upon whether the set is empty or not.

If the c-set is empty, the member and owner record are the same for this set. If the set position requested is LAST or FIRST it is ignored and the owner record (which in this case is also the only member record of the set) is returned as the record found. The positional calls using NEXT or PRIOR, if not proceeded by a LAST or FIRST positional call, are treated in a similar manner. All other positional accesses generate an end of set error status. To tell whether any previous accesses have been made to this set instance a flag in the internal database is turned on and off. This flag is updated appropriately every time a member or owner record of sets with empty c-sets has been accessed. If no error is generated, the database key of the owner record is returned to the main user work area and currency set.

If the c-set is non-empty, all sets in the c-set must be exploded so that all combinations of records are provided to the run-unit. In order to do this, the currency for all sets in the c-set must first be determined. This is done by taking the set currency stored in the main user work area

and using this value as the current set position for the first set in the c-set. The appropriate currencies for all the other sets are determined by doing FIND OWNER accesses on the sets in the c-set in the proper order.

Once the currencies have been set, the positional finds can be performed. A position of LAST or FIRST requires that the FIRST or LAST positional record be found for all sets in the c-set in reverse order. Note that the c-set is a list of sets with the prime order determined by those sets in the unique upward path from the c-record to the original owner of the set. If any of these accesses, other than the first, accesses an empty set, then a FIND NEXT or FIND PRIOR command is executed in the following set in the c-set.

If the positional request is NEXT or PRIOR the proper positional find is made to the first set in the c-set. If an empty set is encountered a FIND NEXT or FIND PRIOR is executed in the following set in the c-set and so forth recursively. When the last set in the c-set is exhausted a non-zero error status is returned to the user.

When the position requested is a number, the proper sequence of FIND LAST, NEXT, FIRST, and PRIOR commands are made had the user not entered a number. For example, if the position requested is 3, the implied FIND FIRST and two FIND NEXT commands are actually executed. When the proper record is found, the database key is used to reset the main user work area currency.

4.5.8 FINDO Command -

The set name for which the command is requested is searched for in CHGDB to see if it has been restructured. If it has not been, the currency for the set is obtained from the main user work area, and a FIND OWNER command executed. The resulting record and database key are used to reset the main user work area currency. If the set has been deleted as a result of restructuring, an error message is returned (see Appendix Number 6).

If the set has been restructured but not deleted, a c-set exists. If the c-set is the empty set (the c-record is also the owner record) there is no database access made and the currency in the main user work area is set using the database key of the c-record.

When the c-set is non-empty, FIND OWNER commands are made for all sets in the c-set. If the owner of the set is already current the FIND OWNER commands are not necessary. Once the owner has been found the currency in the main user work area is reset. Any error that might have occurred is returned to the user.

4.6 Documentation

The routines described in the portion of the paper comprise the implementation as it exists as a result of this paper. The program code and documentation is stored on the

DEC System-10 in several Fortran source code files under user number [4010,54]. Those who are interested in a more detailed look at the routines and those who may want to do further implementation should consult these files.

5.C FURTHUR IMPLEMENTATION AND EXTENSIONS

This section will outline the next steps needed to fully complete the implementation of the dynamic restructuring processsor described in this paper. This work has tried to develop a general implementation strategy and incorporate the data and utility routines to be used by the total implementation. Much thought and work needs to be done to implement the remaining portion of the dynamic restructuring processor.

Section 4.2 outlined the general implementation and testing strategy. The work done for this paper included a detailed implementation of steps 1 and 2. The next step is to implement steps 3 and 4. Step 3 requires the writing and testing of the locator routines. These routines, one for each command to be reconstructed, will search through the most recent generation database and all previous generation databases for the data that is the target of the inputted command.

It is believed that the information stored in the internal database and used by the reconstructor routines is sufficient to perform these locator operations. The locator operations are in many ways the inverse of the reconstructor routines. The reconstructor routines reconstruct the user's database calls to the call or calls that would have been made had he been using the most recent generation schema. The locator routines will take the call in the form needed to access the most recent generation database and change it into the call or calls needed to access the other generation databases. The locator routines must sequence through all coexistent databases to try and locate the target data. These routines must also manage the error messages resulting from its own calls to the databases and the errors returned to the user of the locator routines.

The locator routines can be tested before the final implementation step is taken, i.e. the implementation of the translation routine that transfers data in old generations databases to the most recent generation database. Morgan and Gerritsen's paper [1] gives some insight into exactly how this translation can be accomplished.

The implementation of these final two steps will produce a complete dynamic restructuring processor. The RDDFDP processor to actually compile the restructuring definition language into the necessary files and databases

must also be implemented. This processor can of course be developed in parallel with the other implementation, but the form of the internal database must be totally specified first. The locator routines must at least be conceptually designed and their data requirements ascertained before the interal database can be totally specified.

The RDDFDP processor in general should perform the following operations:

1) Using the RDDL definition and the previous generation schema definition, create the data definition for the next generation schema. The internal database definition is also appended as the first area of all these schema definitions if not done before. The resulting schema definition is outputted for use by applications programmers and also is run through the FDP processor to generate the proper *.SCH file for this generation schema to be used by the restructuring processor.

2) From the RDDL and previous generation schema definition, the c-sets, c-records, etc. are calculated. The relocation paths for data relocation are calculated and tested for uniqueness.

3) The calculated values from step two are then incorporated in the internal database. The database must be augmented by including the data needed to perform translations from the previous generation schema to a new

generation schema. All the data in the CHGDB is stored to allow translation between previous generation schema and the most recent generation schema. The restructuring, however, has modified the previous most recent generation schema. The CHGDB is modified to allow translation to the new most recent generation schema for all previous generations of the schema.

At the present time only the routines FINDAP, FINDPO, FINDC, FINDO, GET, STORE, MODIFY, and DELETE are being reconstructed. Other routines are also candidate for reconstruction, including DELLALL, and FINDV. Several additions and extensions mentioned in the previous sections of this paper are candidates for implementation as well.

6.0 CONCLUSION

The work done for this paper has attempted to show through an implementation, the feasibility of dynamic restructuring of databases. The implementation of a complete dynamic restructuring processor has not been fully attained by this work but brought several steps closer to total implementation.

The completed programming work done for this paper, however, is of importance in its own right. The processes implemented provide a system that will allow several generations of schema to coexist and provide runtime

translation for users or programs using previous generation schema from a database existing under the most recent generation schema definition. This system will remove the common necessity of rewriting applications programs after a conceptual schema restructuring.

The concepts developed by Morgan and Gerritsen [1] have proven adequate for the translations developed for this partial implementation of the dynamic restructuring processor. It is believed that they will also be adequate for the remaining part of the processor to be implemented.

The dynamic restructuring processor, as described in this paper, is only part of any total dynamic restructuring system. Of Socket and Goldberg's [10] stated guidelines for such a system (see Section 2.0 for a listing of their guidelines) the dynamic restructuring processor only addresses guideline 1, the restructuring process must operate correctly, returning the correct data from the database. The issues of deadlock, synchronization of processes, and recovery must be addressed by any total restructuring system that is to be developed from this dynamic restructuring processor. I am confident these problems can be handled.

The final guideline mentioned is that reasonable efficiency must be maintained. No measurement of efficiency has been defined, however, some general comments on efficiency can be made. By necessity the dynamic

restructuring routines require more operating overhead for all database accesses and each access request inputted by the user requires at least one, and often two or more, database accesses. Some restructuring definitions can make the number of database accesses high in number reducing operating efficiency.

There is no way at this point to measure the operating efficiency of the dynamic restructuring processor because further implementation is necessary before a total processor exsists.

As the introduction of this paper indicates, dynamic restructuring is of particular importance when database applications require complete and uninteruptable database integrity. Dynamic restructuring also may be applicable to restructuring very large databases, especially those that are highly volatile, i.e. the number of record deletions and replacements is high in relation to the number of record accesses. Dynamic restructuring does not require data in prior generation databases to be restructured to the present generation database before it is deleted or replaced.

To determine the tradeoffs between the two types of restructuring options (traditional or offline restructuring versus dynamic restructuring) consider the following analysis of restructuring costs. There are two types of cost associated with restructuring. The cost of providing translation back to the user's generation schema would be

the same despite the type of restructuring used therefore it is irrelevant here.

The other type of cost is the cost of actually performing the database restructuring. Traditional restructuring has a large fixed processing cost, although the average cost per record may be relatively low. Dynamic restructuring has a variable type cost. The independent variable upon which this cost is dependent is the number of records of previous generation databases initially accessed (not replaced or deleted). This is the independent variable because the restructuring processing cost is a one time cost incurred when records in prior generation databases are first accessed. A low fixed cost for the storage needed by the dynamic restructuring processor also exists. The restructuring costs as defined above can now be compared using the usual break-even analysis. The traditional or offline restructuring costs are fixed and do not vary with the independent variable defined before. The dynamic restructuring processor has a high variable and low fixed cost in relation to the independent variable. The break-even point exists and can probably be calculated if the correct costs are known.

The independent variable value for any particular database is dependent upon a number of things including size of the database. and volatility of the database. The independent variable can usually be measured by some

probabilistic measure. Some databases will fall below the break-even point and thus it would be cost effective to use dynamic restructuring in these situations.

It is impossible at this point to measure the restructuring costs of dynamic restructuring because the entire processor has not been implemented. When implementation is completed such costs can be determined and an analysis made as to the proper break-even point and what databases are above and below the point can be determined. This analysis will indicate those databases where dynamic restructuring is a less costly technique for restructuring.

BIBLIOGRAPHY

1. Gerritsen, Rob, and Morgan, Howard L., "Dynamic Restructuring of Databases with Generation Data Structures", Dept Decision Sciences, Working Paper 75-12-02, The Wharton School, December 1975.

2. CODASYL, CODASYL Data Base Task Group April 71 Report, available from ACM, New York City.

3. Shu, Nan C., Barron C. Housel, and Vincent Y. Lum, "Convert: A High level Translation Definition Language for Data Conversion," Comm. ACM 18 10, October 1975, pp557-567.

4. Merten, Alan G. and James P. Fry, "A Data Description Language Approach to File Translation," Proceedins, ACM SIGMOD Workchop on Data Descriptin Access and Control, May 1974, pp 191-205.

5. Ramirez, J. A., N. A. Kin, and K. S. Prywes, "Automatic Generation of Data Conversion Programs Using a Data Description Language," Proceedings, ACM SIGMOD Workshop on Data Description Access and Control, May 1974, pp207-225.

6. Burk, J. R., "DMS Data Base Restructuring," Xerox Technology Report (Work Order F34234), May 1971.

7. Gerritsen, Rob, Howard L. Morgan, and Micheal D. Zisman, "On some Metrics for Databases, or What is a Very Large Database?," Decision Sciences Working Paper 76-04-08, April 1976.

8. Gerritsen, Rob, Ricardo Cortes, Jim Ribeiro, and Ruth Zowader, "WAND User's Guide," Decision Sciences Working Paper 76-01-03, April 1976.

9. Winslow, L.E. and Lee, J. C., "Optimal Choice of Restructuring Points," Proceedings of the International Conference on Very Large Databases, September 1975.

10. Socket, Gary H., Goldberg, Robert P., "Motivation for Database Reorganization Performed Concurrently with Usage," Working Paper TR 16-76, Aiken Computing Labratory, Harvard University, Cambridge, Mass.

11. ANSI/SPARC, "Study Group on Database Management Systems - Interim Report," 75-02-08, American National Standards Institute, Washingdon, D. C.

APPENDIX NUMBER 1
EXAMPLE MEDICAL DATA BASE
BEFORE ANY RESTRUCTURING

```
SCHEMA NAME IS MEDICALDB.

AREA NAME IS MEDICAL.

RECORD NAME IS HOSPITAL
    LOCATION MODE IS CALC USING HOSPNAME
        DUPLICATES NOT ALLOWED
    HOSPNAME        TYPE IS CHARACTER 20.

RECORD NAME IS DOCTOR
    LOCTATION MODE IS CALC USING DOCNAME
        DUPLICATES NOT ALLOWED
    DOCNAME         TYPE IS CHARACTER 30
    DOCADDRESS      TYPE IS CHARACTER 30.

SET NAME IS HOSDOC
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS HOSPITAL
    MEMBER IS DOCTOR.

RECORD NAME IS PATIENT
    LOCATION MODE IS CALC USING PATNAME
        DUPLICATES NOT ALLOWED
    PATNAME         TYPE IS CHARACTER 30
    PATAGE          TYPE IS FIXED
    TREATMENT       TYPE IS CHARACTER 40.

SET NAME IS DOCPAT
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS DOCTOR
    MEMBER IS PATIENT.

RECORD NAME IS WING
    LOCATION MODE IS CALC USING WINGNAME
        DUPLICATES NOT ALLOWED
    WINGNAME        TYPE IS CHARACTER 30
    WINGSIZE        TYPE IS FIXED.

SET NAME IS HOSWING
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS HOSPITAL
    MEMBER IS WING.

RECORD NAME IS NURSE
    LOCATION MODE IS CALC USING NURSNAME
```

```
        DUPLICATES ALLOWED
    NURSNAME        TYPE IS CHARACTER 30
    NURSTYPE        TYPE IS CHARACTER 10.

SET NAME IS ASSGMT
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS WING
    MEMBER IS NURSE.
```

APPENDIX NUMBER 2
RESTRUCTURING DEFINITION

```
CHANGE NAME OF PATIENT RECORD TO DIAGNOSIS.

INCORPORATE RECORD NAME IS PATIENT
    LOCATION MODE IS CALC USING PATID
        DUPLICATES NOT ALLOWED
    PATID TYPE IS FIXED.

INCORPORATE SET NAME IS PATDOC
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS PATIENT
    MEMBER IS DIAGNOSIS.

RELOCATE PATNAME OF DIAGNOSIS IN PATIENT
    USING UNIQUE PATH.

RELOCATE PATAGE OF DIAGNOSIS IN PATIENT
    USING PATDOC PATH.

INCORPORATE RECORD NAME IS FLOOR
    LOCATION MODE IS CALC USING FLRNAME
        DUPLICATES ALLOWED
    FLRNAME          TYPE IS CHARACTER 15
    FLRSIZE          TYPE IS FIXED.

CHANGE NAME OF SET ASSGMT TO WINGFLR.

CHANGE SET NAME IS WINGFLR
    MEMBER IS RETROACTIVE FLOOR
    SET OCCURANCE SELECTION IS THRU LOCATION MODE OF OWNER
    USING FLRNAME.

EXCISE MEMBER NURSE FROM SET WINGFLR.

INCORPORATE SET NAME IS ASSGMT
    MODE IS CHAIN
    ORDER IS FIRST
    OWNER IS FLOOR
    MEMBER IS NURSE.
```

APPENDIX NUMBER 3
SCHEMA FOR THE CHANGE DATA BASE

```
AREA NAME IS DBCHG.

RECORD NAME IS SCHMA
    LOCATION MODE IS CALC USING NAMSCH
        DUPLICATES NOT ALLOWED
    NAMSCH      TYPE IS CHARACTER 10
    NUMGEN      TYPE IS FIXED.

RECORD NAME IS GENCHG
    LOCATION MODE IS VIA GENERATNS
    GENNUM      TYPE IS FIXED
    GSCHNAM     TYPE IS CHARACTER 10.

RECORD NAME IS SETCHG
    LOCATION MODE IS VIA CHGSET
    SOLDNAM     TYPE IS CHARACTER 10
    SCHGFLG     TYPE IS FIXED.

RECORD NAME IS LSTMEMBER
    LOCATION MODE IS VIA LIST
    SETNAME     TYPE IS CHARACTER 10.

RECORD NAME IS RECCHG
    LOCATION MODE IS VIA CHGREC
    ROLDNAM     TYPE IS CHARACTER 10
    RCRECORD    TYPE IS CHARACTER 10
    RCHGFLG     TYPE IS FIXED.

RECORD NAME IS ITMCHG
    LOCATION MODE IS VIA CHGITM
    IOLDNAM     TYPE IS CHARACTER 10
    ICHGFLG     TYPE IS FIXED
    ICRECORD    TYPE IS CHARACTER 10.

RECORD NAME IS PATHMEM
    LOCATION MODE IS VIA PATH
    SNAME TYPE IS CHARACTER 10.

SET NAME IS GENERATNS
    MODE IS CHAIN
    ORDER IS SORTED
    OWNER IS SCHMA
    MEMBER IS GENCHG
      ASCENDING KEY IS GENNUM.

SET NAME IS CHGSET
    MODE IS CHAIN
    ORDER IS SORTED
    OWNER IS GENCHG
```

```
    MEMBER IS SETCHG
      ASCENDING KEY IS SOLDNAM.

SET NAME IS CHGREC
    MODE IS CHAIN
    ORDER IS SORTED
    OWNER IS GENCHG
    MEMBER IS RECCHG
      ASCENDING KEY IS ROLDNAM.

SET NAME IS CHGITM
    MODE IS CHAIN
    ORDER IS SORTED
    OWNER IS GENCHG
    MEMBER IS ITMCHG
      ASCENDING KEY IS IOLDNAM.

SET NAME IS LIST
    MODE IS CHAIN
    LINKED TO PRIOR
    ORDER IS NEXT
    OWNER IS SETCHG
    MEMBER IS LSTMEMBR.

SET NAME IS PATH
    MODE IS CHAIN
    LINKED TO PRIOR
    ORDER IS NEXT
    OWNER IS ITMCHG
    MEMBER IS PATHMEM.
```

APPENDIX NUMBER 4
WAND SCHEMA DDL

SYMBOL	MEANING
___ (underline)	WORD MUST APPEAR
( )	PHRASE MAY BE OMITTED
[ ]	ONLY ONE OF THE LINES MAY BE USED

Lower case words must be replaced by a user-defined name or value.

```
SCHEMA NAME IS schema-name
        (PRIVACY LOCK IS integer PAGES)
        (DATABASE SIZE IS integer PAGES)
        (PAGE SIZE IS integer WORDS).

AREA NAME IS area-name
        (AREA SIZE IS integer PAGES)
        (PAGE SIZE IS integer WORDS).

RECORD NAME IS record-name
   LOCATION MODE IS
        [VIA set-name                                                    ]
        [CALC USING item-name-1[DUPLICATES ARE (NOT) ALLOWED]
        [DIRECT                                                          ]

   (WITHIN area-name).

item-name-2 TYPE IS
        [CHARACTER integer]
        [FIXED            ]
        [REAL             ].

SET NAME IS set-name
        MODE IS CHAIN
                (LINKED TO PRIOR)

        ORDER IS
                [FIRST ]
                [LAST  ]
                [NEXT  ]
                [PRIOR ]
                [SORTED]

        OWNER IS record-name-1

        MEMBER IS record-name-2
                (LINKED TO OWNER)

          ([ASCENDING ] KEY IS item-name-1)
           [DESCENDING]
```

```
(SET OCCURANCE SELECTION IS THRU
      [CURRENT OF SET          ]
      [LOCATION MODE OF OWNER]
        (ALIAS FOR item-name-2 IS data-name)  ).
```

APPENDIX NUMBER 5
RESTRUCTURING DATA DEFINITION LANGUAGE

SYMBOL	MEANING
___ (underline)	WORD MUST APPEAR
( )	PHRASE MAY BE OMITTED
[ ]	ONLY ONE OF THESE LINES MAY BE USED

Lower case words must be replaced by a user-defined name or value.

```
INCORPORATE  [area-entry.                                          ]
             [record-entry.                                        ]
             [set-entry.                                           ]
             [IN record-name-1 RECORD data-sub-entry.              ]
             [IN set-name-1 SET member-sub-entry.                  ]

EXCISE       [AREA NAMED area-name-1.                              ]
             [RECORD NAMED record-name-2.                          ]
             [SET NAMED set-name-2.                                ]
             [item-name-1 FROM record-name-3 RECORD.               ]
             [MEMBER NAMED record-name-4 FROM set-name-3 SET.      ]

RELOCATE  (level-number) database-data-name
          OF record-name-5 RECORD
              IN record-name-6 RECORD
          USING [UNIQUE                 ]
                [set-1(,set- ...)] PATH.

CHANGE SET NAMED set-name-1
          MEMBER IS [RETROACTIVE] record-name-1
              (SET OCCURANCE SELECTION IF THRU
                  LOCATION MODE OF OWNER USING
                      fully-qualified-data-name-from-old-schema
                          FOR data-base-identifier ) .
```

APPENDIX NUMBER 6
RESTRUCTURING ERROR CODES

63 - An error in using an internal stack was encountered.

1260 - Because of restructuring the STORE command cannot be performed.

6010 - Record or set no longer exists because of restructuring.

6012 - Unable to reconstruct calc access for this record due to restructuring.

6020 - Error in reconstructing a record from its c-record.

Any unexpected error encountered in making accesses to the internal database has 7000 added to it and is returned.
Note: This convention was used for ease of debugging, a more comprehensive error may be used later.